In [2]:
```
# reference : https://github.com/keras-team/keras-io/blob/master/examples/graph/i
```

In [3]:
```
import os
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

In [4]:
```
# Dataset - Cora
# The dataset has two tap-separated files: cora.cites and cora.content.
# (i)The cora.cites includes the citation records with two columns:
#    cited_paper_id (target) and citing_paper_id (source).
# (ii)The cora.content includes the paper content records
#     with 1,435 columns: paper_id, subject, and 1,433 binary features.
```

In [5]:
```
# get dataset
zip_file = keras.utils.get_file(
    fname="cora.tgz",
    origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",
    extract=True,
)
data_dir = os.path.join(os.path.dirname(zip_file), "cora")
```

In [6]:
```
# Data exploration
```

In [7]:
```
citations = pd.read_csv(
    os.path.join(data_dir, "cora.cites"),
    sep="\t",
    header=None,
    names=["target", "source"],
)
print("Citations shape:", citations.shape)
```

```
Citations shape: (5429, 2)
```

In [8]:
```
citations.sample(frac=1).head()

#The target column includes the paper ids cited by the paper ids
#in the source column
```

Out[8]:

	target	source
934	3243	1104055
3308	54132	1128985
3881	94639	593813
2503	26850	1114605
1599	8703	502574

In [9]:
```
column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] + ["subject'
papers = pd.read_csv(
    os.path.join(data_dir, "cora.content"), sep="\t", header=None, names=column_n
)
print("Papers shape:", papers.shape)
```

```
Papers shape: (2708, 1435)
```

In [10]:
```
print(papers.sample(5).T)

# The DataFrame includes the paper_id and the subject columns,
# as well as 1,433 binary column representing whether a term exists
# in the paper or not.
```

```
                535      1986        1976   2327                    725
paper_id    1104007    167656     1121176  62274                 126793
term_0            0         0           0      0                      0
term_1            0         0           0      0                      0
term_2            1         0           0      0                      0
term_3            0         0           0      0                      0
...             ...       ...         ...    ...                    ...
term_1429         0         0           0      0                      0
term_1430         0         0           0      0                      0
term_1431         0         0           0      0                      0
term_1432         0         0           0      0                      0
subject      Theory    Theory  Case_Based Theory  Probabilistic_Methods

[1435 rows x 5 columns]
```

In [11]:
```
print(papers.subject.value_counts())

# 7 classes/subjects and count of papers in each subject
```

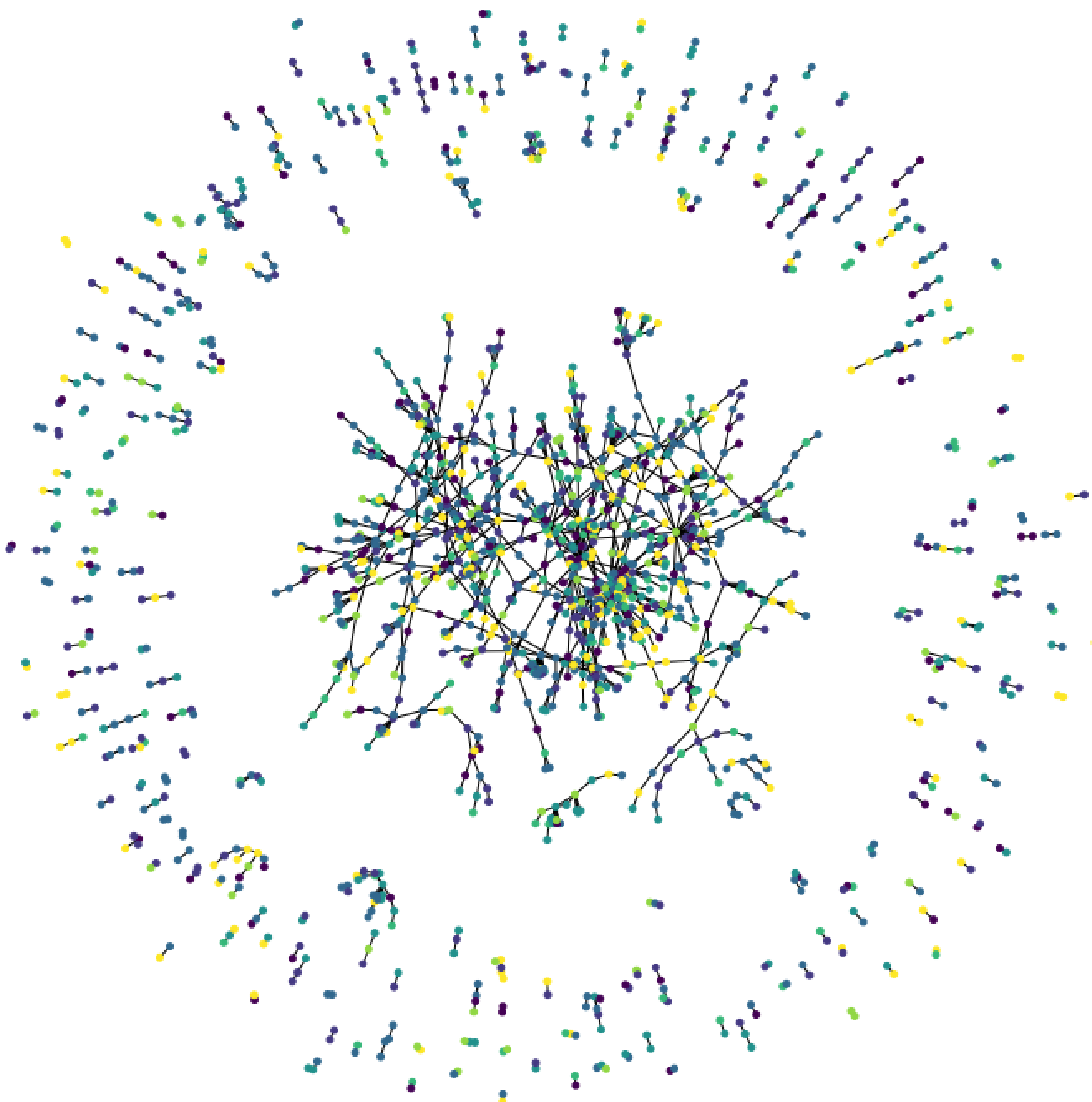
```
Neural_Networks           818
Probabilistic_Methods     426
Genetic_Algorithms        418
Theory                    351
Case_Based                298
Reinforcement_Learning    217
Rule_Learning             180
Name: subject, dtype: int64
```

In [12]:
```
# convert to zero based indices
class_values = sorted(papers["subject"].unique())
class_idx = {name: id for id, name in enumerate(class_values)}
paper_idx = {name: idx for idx, name in enumerate(sorted(papers["paper_id"].uniqu

papers["paper_id"] = papers["paper_id"].apply(lambda name: paper_idx[name])
citations["source"] = citations["source"].apply(lambda name: paper_idx[name])
citations["target"] = citations["target"].apply(lambda name: paper_idx[name])
papers["subject"] = papers["subject"].apply(lambda value: class_idx[value])
```

In [13]:
```
plt.figure(figsize=(10, 10))
colors = papers["subject"].tolist()
cora_graph = nx.from_pandas_edgelist(citations.sample(n=1500))
subjects = list(papers[papers["paper_id"].isin(list(cora_graph.nodes))]["subject'
nx.draw_spring(cora_graph, node_size=15, node_color=subjects)

# each node is a paper, each color designates a specific subject
```



In [14]:
```
train_data, test_data = [], []

for _, group_data in papers.groupby("subject"):
    # Select around 50% of the dataset for training.
    random_selection = np.random.rand(len(group_data.index)) <= 0.5
    train_data.append(group_data[random_selection])
    test_data.append(group_data[~random_selection])

train_data = pd.concat(train_data).sample(frac=1)
test_data = pd.concat(test_data).sample(frac=1)

print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
```

```
Train data shape: (1336, 1435)
Test data shape: (1372, 1435)
```

In [15]:
```
hidden_units = [32, 32]
learning_rate = 0.01
dropout_rate = 0.5
num_epochs = 300
batch_size = 256
```

In [16]:
```
def run_experiment(model, x_train, y_train):
    # Compile the model.
    # Optimizer - Adam
    # loss criterion - Sparse Categorical Cross entropy
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],
    )
    # Create an early stopping callback.
    early_stopping = keras.callbacks.EarlyStopping(
        monitor="val_acc", patience=50, restore_best_weights=True
    )
    # Fit the model.
    history = model.fit(
        x=x_train,
        y=y_train,
        epochs=num_epochs,
        batch_size=batch_size,
        validation_split=0.15,
        callbacks=[early_stopping],
    )

    return history
```

In [17]:
```
# helper function to visualize
def display_learning_curves(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    ax1.plot(history.history["loss"])
    ax1.plot(history.history["val_loss"])
    ax1.legend(["train", "test"], loc="upper right")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")

    ax2.plot(history.history["acc"])
    ax2.plot(history.history["val_acc"])
    ax2.legend(["train", "test"], loc="upper right")
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Accuracy")
    plt.show()
```

In [18]:
```
# This feed forward nn is used in the baseline and the GNN models
def create_ffn(hidden_units, dropout_rate, name=None):
    fnn_layers = []

    for units in hidden_units:
        fnn_layers.append(layers.BatchNormalization())
        fnn_layers.append(layers.Dropout(dropout_rate))
        fnn_layers.append(layers.Dense(units, activation=tf.nn.gelu))

    return keras.Sequential(fnn_layers, name=name)
```

In [19]:
```
feature_names = set(papers.columns) - {"paper_id", "subject"}
num_features = len(feature_names)
num_classes = len(class_idx)

# Create train and test features as a numpy array.
x_train = train_data[feature_names].to_numpy()
x_test = test_data[feature_names].to_numpy()
# Create train and test targets as a numpy array.
y_train = train_data["subject"]
y_test = test_data["subject"]
```

In [20]:
```
def create_baseline_model(hidden_units, num_classes, dropout_rate=0.2):
    inputs = layers.Input(shape=(num_features,), name="input_features")
    x = create_ffn(hidden_units, dropout_rate, name=f"ffn_block1")(inputs)
    for block_idx in range(4):
        # Create an FFN block.
        x1 = create_ffn(hidden_units, dropout_rate, name=f"ffn_block{block_idx +
        # Add skip connection.
        x = layers.Add(name=f"skip_connection{block_idx + 2}")([x, x1])
    # Compute logits.
    logits = layers.Dense(num_classes, name="logits")(x)
    # Create the model.
    return keras.Model(inputs=inputs, outputs=logits, name="baseline")


baseline_model = create_baseline_model(hidden_units, num_classes, dropout_rate)
baseline_model.summary()
```

```
Model: "baseline"
________________________________________________________________________________
___________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
===================
input_features (InputLayer)     [(None, 1433)]       0
________________________________________________________________________________
___________________
ffn_block1 (Sequential)         (None, 32)           52804       input_features
[0][0]
________________________________________________________________________________
___________________
ffn_block2 (Sequential)         (None, 32)           2368        ffn_block1[0]
[0]
________________________________________________________________________________
___________________
skip_connection2 (Add)          (None, 32)           0           ffn_block1[0]
[0]
                                                                 ffn_block2[0]
[0]
________________________________________________________________________________
___________________
ffn_block3 (Sequential)         (None, 32)           2368        skip_connectio
n2[0][0]
________________________________________________________________________________
___________________
skip_connection3 (Add)          (None, 32)           0           skip_connectio
n2[0][0]
                                                                 ffn_block3[0]
[0]
________________________________________________________________________________
___________________
ffn_block4 (Sequential)         (None, 32)           2368        skip_connectio
n3[0][0]
________________________________________________________________________________
___________________
skip_connection4 (Add)          (None, 32)           0           skip_connectio
n3[0][0]
                                                                 ffn_block4[0]
```

```
[0]
________________________________________________________________________________
__________________
ffn_block5 (Sequential)         (None, 32)          2368        skip_connectio
n4[0][0]
________________________________________________________________________________
__________________
skip_connection5 (Add)          (None, 32)          0           skip_connectio
n4[0][0]
                                                                ffn_block5[0]
[0]
________________________________________________________________________________
__________________
logits (Dense)                  (None, 7)           231         skip_connectio
n5[0][0]
================================================================================
==================
Total params: 62,507
Trainable params: 59,065
Non-trainable params: 3,442
________________________________________________________________________________
__________________
```

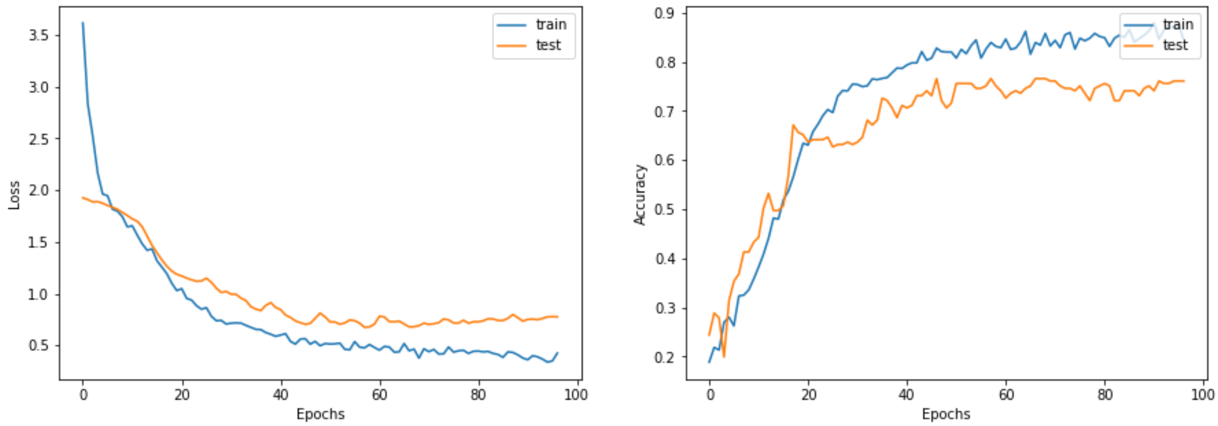
In [21]:

```
#Train the baseline model

history = run_experiment(baseline_model, x_train, y_train)
```

```
Epoch 1/300
5/5 [==============================] - 2s 74ms/step - loss: 3.6149 - acc: 0.1
885 - val_loss: 1.9251 - val_acc: 0.2438
Epoch 2/300
5/5 [==============================] - 0s 11ms/step - loss: 2.8318 - acc: 0.2
185 - val_loss: 1.9094 - val_acc: 0.2886
Epoch 3/300
5/5 [==============================] - 0s 11ms/step - loss: 2.5213 - acc: 0.2
132 - val_loss: 1.8861 - val_acc: 0.2786
Epoch 4/300
5/5 [==============================] - 0s 11ms/step - loss: 2.1653 - acc: 0.2
696 - val_loss: 1.8888 - val_acc: 0.1990
Epoch 5/300
5/5 [==============================] - 0s 11ms/step - loss: 1.9623 - acc: 0.2
802 - val_loss: 1.8737 - val_acc: 0.3134
Epoch 6/300
5/5 [==============================] - 0s 11ms/step - loss: 1.9457 - acc: 0.2
626 - val_loss: 1.8521 - val_acc: 0.3532
Epoch 7/300
```

In [22]:
```
display_learning_curves(history)
```



In [23]:
```
_, test_accuracy = baseline_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

```
Test accuracy: 74.42%
```

In [24]:
```
# used to create new instances for testing the baseline model
def generate_random_instances(num_instances):
    token_probability = x_train.mean(axis=0)
    instances = []
    for _ in range(num_instances):
        probabilities = np.random.uniform(size=len(token_probability))
        instance = (probabilities <= token_probability).astype(int)
        instances.append(instance)

    return np.array(instances)


def display_class_probabilities(probabilities):
    for instance_idx, probs in enumerate(probabilities):
        print(f"Instance {instance_idx + 1}:")
        for class_idx, prob in enumerate(probs):
            print(f"- {class_values[class_idx]}: {round(prob * 100, 2)}%")
```

In [25]:
```
new_instances = generate_random_instances(num_classes)
logits = baseline_model.predict(new_instances)
probabilities = keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
Instance 1:
- Case_Based: 0.66%
- Genetic_Algorithms: 6.26%
- Neural_Networks: 77.08%
- Probabilistic_Methods: 11.92%
- Reinforcement_Learning: 1.25%
- Rule_Learning: 0.44%
- Theory: 2.39%
Instance 2:
- Case_Based: 3.08%
- Genetic_Algorithms: 3.94%
- Neural_Networks: 69.44%
- Probabilistic_Methods: 14.55%
- Reinforcement_Learning: 1.5%
- Rule_Learning: 1.61%
- Theory: 5.88%
Instance 3:
- Case_Based: 0.61%
- Genetic_Algorithms: 46.42%
- Neural_Networks: 24.74%
- Probabilistic_Methods: 12.26%
- Reinforcement_Learning: 11.35%
- Rule_Learning: 0.72%
- Theory: 3.9%
Instance 4:
- Case_Based: 15.17%
- Genetic_Algorithms: 48.97%
- Neural_Networks: 5.29%
- Probabilistic_Methods: 16.96%
- Reinforcement_Learning: 7.28%
- Rule_Learning: 0.51%
- Theory: 5.82%
Instance 5:
- Case_Based: 1.6%
- Genetic_Algorithms: 2.94%
- Neural_Networks: 19.61%
- Probabilistic_Methods: 27.16%
- Reinforcement_Learning: 0.68%
- Rule_Learning: 10.02%
- Theory: 37.99%
Instance 6:
- Case_Based: 7.16%
- Genetic_Algorithms: 33.34%
- Neural_Networks: 17.04%
- Probabilistic_Methods: 12.98%
- Reinforcement_Learning: 6.08%
- Rule_Learning: 7.61%
- Theory: 15.79%
Instance 7:
- Case_Based: 1.23%
- Genetic_Algorithms: 37.76%
- Neural_Networks: 36.22%
```

```
- Probabilistic_Methods: 4.96%
- Reinforcement_Learning: 17.31%
- Rule_Learning: 1.44%
- Theory: 1.09%
```

In [26]:
```
# Graph preparation for loading into a GNN is hard
# Below approach assumes entire graph fits in memory -
```

In [27]:
```
# Create an edges array (sparse adjacency matrix) of shape [2, num_edges].
edges = citations[["source", "target"]].to_numpy().T

# Create an edge weights array of ones. - not used here since the citations are r
edge_weights = tf.ones(shape=edges.shape[1])

# Create a node features array of shape [num_nodes, num_features].
# Nodes are the papers and node_features word presence binary vectors for each pa
node_features = tf.cast(
    papers.sort_values("paper_id")[feature_names].to_numpy(), dtype=tf.dtypes.flo
)

# Create graph info tuple with node_features, edges, and edge_weights.
graph_info = (node_features, edges, edge_weights)

print("Edges shape:", edges.shape)
print("Nodes shape:", node_features.shape)
```

```
Edges shape: (2, 5429)
Nodes shape: (2708, 1433)
```

In [28]:
```python
# The GNN
class GraphConvLayer(layers.Layer):
    def __init__(
        self,
        hidden_units,
        dropout_rate=0.2,
        aggregation_type="mean",
        combination_type="concat",
        normalize=False,
        *args,
        **kwargs,
    ):
        super(GraphConvLayer, self).__init__(*args, **kwargs)

        self.aggregation_type = aggregation_type
        self.combination_type = combination_type
        self.normalize = normalize

        self.ffn_prepare = create_ffn(hidden_units, dropout_rate)
        if self.combination_type == "gated":
            self.update_fn = layers.GRU(
                units=hidden_units,
                activation="tanh",
                recurrent_activation="sigmoid",
                dropout=dropout_rate,
                return_state=True,
                recurrent_dropout=dropout_rate,
            )
        else:
            self.update_fn = create_ffn(hidden_units, dropout_rate)

    # input node representations are processed using a FFN to produce a message
    def prepare(self, node_repesentations, weights=None):
        # node_repesentations shape is [num_edges, embedding_dim].
        messages = self.ffn_prepare(node_repesentations)
        if weights is not None:
            messages = messages * tf.expand_dims(weights, -1)
        return messages

    # messages of the neighbours of each node are aggregated using operations suc
    def aggregate(self, node_indices, neighbour_messages):
        # node_indices shape is [num_edges].
        # neighbour_messages shape: [num_edges, representation_dim].
        num_nodes = tf.math.reduce_max(node_indices) + 1
        if self.aggregation_type == "sum":
            aggregated_message = tf.math.unsorted_segment_sum(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "mean":
            aggregated_message = tf.math.unsorted_segment_mean(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "max":
            aggregated_message = tf.math.unsorted_segment_max(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
```

```
        else:
            raise ValueError(f"Invalid aggregation type: {self.aggregation_type}.

        return aggregated_message

    # node representations and aggregated messages are updated on the current noc
    def update(self, node_repesentations, aggregated_messages):
        # node_repesentations shape is [num_nodes, representation_dim].
        # aggregated_messages shape is [num_nodes, representation_dim].
        if self.combination_type == "gru":
            # Create a sequence of two elements for the GRU layer.
            h = tf.stack([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "concat":
            # Concatenate the node_repesentations and aggregated_messages.
            h = tf.concat([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "add":
            # Add node_repesentations and aggregated_messages.
            h = node_repesentations + aggregated_messages
        else:
            raise ValueError(f"Invalid combination type: {self.combination_type}.

        # Apply the processing function.
        node_embeddings = self.update_fn(h)
        if self.combination_type == "gru":
            node_embeddings = tf.unstack(node_embeddings, axis=1)[-1]

        if self.normalize:
            node_embeddings = tf.nn.l2_normalize(node_embeddings, axis=-1)
        return node_embeddings

    def call(self, inputs):
        """Process the inputs to produce the node_embeddings.

        inputs: a tuple of three elements: node_repesentations, edges, edge_weigh
        Returns: node_embeddings of shape [num_nodes, representation_dim].
        """

        node_repesentations, edges, edge_weights = inputs
        # Get node_indices (source) and neighbour_indices (target) from edges.
        node_indices, neighbour_indices = edges[0], edges[1]
        # neighbour_repesentations shape is [num_edges, representation_dim].
        neighbour_repesentations = tf.gather(node_repesentations, neighbour_indic

        # Prepare the messages of the neighbours.
        neighbour_messages = self.prepare(neighbour_repesentations, edge_weights)
        # Aggregate the neighbour messages.
        aggregated_messages = self.aggregate(node_indices, neighbour_messages)
        # Update the node embedding with the neighbour messages.
        return self.update(node_repesentations, aggregated_messages)
```

In [29]:
```
# The classifier
# Approach : reference : https://arxiv.org/abs/2011.08843 (Design Space for GNN)
# Apply preprocessing using FFN to the node features to generate initial node rep
# Apply one or more graph convolutional layer, with skip connections, to the node
# Apply post-processing using FFN to the node embeddings to generat the final noc
# Feed the node embeddings in a Softmax layer to predict the node class.

class GNNNodeClassifier(tf.keras.Model):
    def __init__(
        self,
        graph_info,
        num_classes,
        hidden_units,
        aggregation_type="sum",
        combination_type="concat",
        dropout_rate=0.2,
        normalize=True,
        *args,
        **kwargs,
    ):
        super(GNNNodeClassifier, self).__init__(*args, **kwargs)

        # Unpack graph_info to three elements: node_features, edges, and edge_wei
        node_features, edges, edge_weights = graph_info
        self.node_features = node_features
        self.edges = edges
        self.edge_weights = edge_weights
        # Set edge_weights to ones if not provided.
        if self.edge_weights is None:
            self.edge_weights = tf.ones(shape=edges.shape[1])
        # Scale edge_weights to sum to 1.
        self.edge_weights = self.edge_weights / tf.math.reduce_sum(self.edge_weig

        # Create a process layer.
        self.preprocess = create_ffn(hidden_units, dropout_rate, name="preprocess
        # Create the first GraphConv layer.
        self.conv1 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv1",
        )
        # Create the second GraphConv layer.
        self.conv2 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv2",
        )
        # Create a postprocess layer.
        self.postprocess = create_ffn(hidden_units, dropout_rate, name="postproce
        # Create a compute logits layer.
```

```
        self.compute_logits = layers.Dense(units=num_classes, name="logits")

    def call(self, input_node_indices):
        # Preprocess the node_features to produce node representations.
        x = self.preprocess(self.node_features)
        # Apply the first graph conv layer.
        x1 = self.conv1((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x1 + x
        # Apply the second graph conv layer.
        x2 = self.conv2((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x2 + x
        # Postprocess node embedding.
        x = self.postprocess(x)
        # Fetch node embeddings for the input node_indices.
        node_embeddings = tf.squeeze(tf.gather(x, input_node_indices))
        # Compute logits
        return self.compute_logits(node_embeddings)
```

In [30]:
```
gnn_model = GNNNodeClassifier(
    graph_info=graph_info,
    num_classes=num_classes,
    hidden_units=hidden_units,
    dropout_rate=dropout_rate,
    name="gnn_model",
)

print("GNN output shape:", gnn_model([1, 10, 100]))

gnn_model.summary()
```

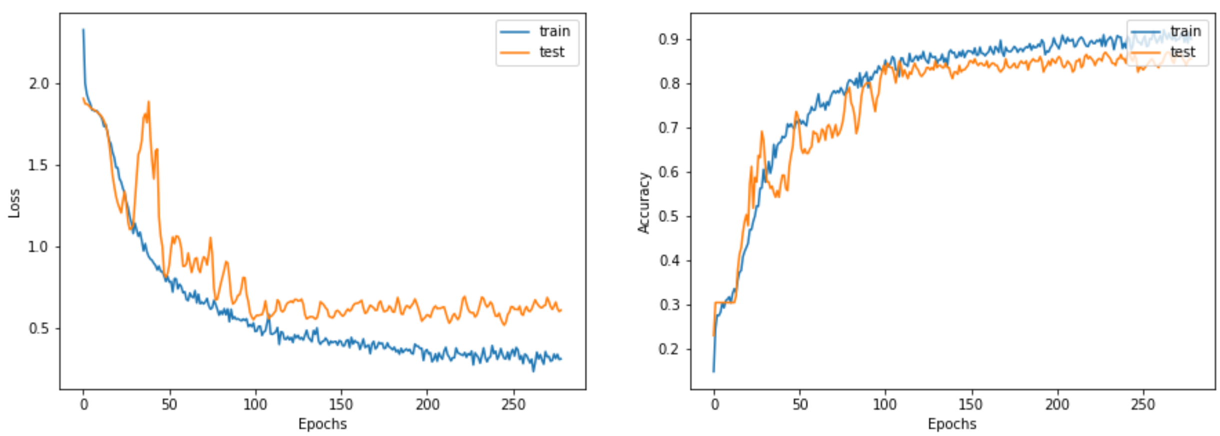
```
GNN output shape: tf.Tensor(
[[-0.24259529  0.13979076 -0.05243582  0.03642052 -0.13590422 -0.15124497
   0.0684088 ]
 [ 0.00158797  0.08392508 -0.24299659  0.03110595 -0.16308787 -0.17948398
   0.21690385]
 [-0.22076407  0.20299804 -0.08488648  0.08628957 -0.07223565  0.00330497
   0.11043227]], shape=(3, 7), dtype=float32)
Model: "gnn_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
preprocess (Sequential)      (2708, 32)                52804
_________________________________________________________________
graph_conv1 (GraphConvLayer) multiple                  5888
_________________________________________________________________
graph_conv2 (GraphConvLayer) multiple                  5888
_________________________________________________________________
postprocess (Sequential)     (2708, 32)                2368
_________________________________________________________________
logits (Dense)               multiple                  231
=================================================================
Total params: 67,179
Trainable params: 63,481
Non-trainable params: 3,698
_________________________________________________________________
```

In [31]:
```python
#Train the GNN model

x_train = train_data.paper_id.to_numpy()
history = run_experiment(gnn_model, x_train, y_train)
```

```
Epoch 1/300
WARNING:tensorflow:From c:\zzz_tools\python\lib\site-packages\tensorflow\pyth
on\ops\array_ops.py:5049: calling gather (from tensorflow.python.ops.array_op
s) with validate_indices is deprecated and will be removed in a future versio
n.
Instructions for updating:
The `validate_indices` argument has no effect. Indices are always validated o
n CPU and never validated on GPU.
5/5 [==============================] - 3s 152ms/step - loss: 2.3290 - acc: 0.
1471 - val_loss: 1.9074 - val_acc: 0.2289
Epoch 2/300
5/5 [==============================] - 0s 63ms/step - loss: 2.0018 - acc: 0.2
441 - val_loss: 1.8755 - val_acc: 0.3035
Epoch 3/300
5/5 [==============================] - 0s 64ms/step - loss: 1.9345 - acc: 0.2
749 - val_loss: 1.8738 - val_acc: 0.3035
Epoch 4/300
5/5 [==============================] - 0s 65ms/step - loss: 1.8992 - acc: 0.2
740 - val_loss: 1.8663 - val_acc: 0.3035
```

In [32]:
```python
display_learning_curves(history)
```



In [33]:
```python
x_test = test_data.paper_id.to_numpy()
_, test_accuracy = gnn_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

```
Test accuracy: 81.63%
```

In [34]:
```
# Add new instances to test GNN model against these

# First we add the N new_instances as nodes to the graph
# by appending the new_instance to node_features.
num_nodes = node_features.shape[0]
new_node_features = np.concatenate([node_features, new_instances])
# Second we add the M edges (citations) from each new node to a set
# of existing nodes in a particular subject
new_node_indices = [i + num_nodes for i in range(num_classes)]
new_citations = []
for subject_idx, group in papers.groupby("subject"):
    subject_papers = list(group.paper_id)
    # Select random x papers specific subject.
    selected_paper_indices1 = np.random.choice(subject_papers, 5)
    # Select random y papers from any subject (where y < x).
    selected_paper_indices2 = np.random.choice(list(papers.paper_id), 2)
    # Merge the selected paper indices.
    selected_paper_indices = np.concatenate(
        [selected_paper_indices1, selected_paper_indices2], axis=0
    )
    # Create edges between a citing paper idx and the selected cited papers.
    citing_paper_indx = new_node_indices[subject_idx]
    for cited_paper_idx in selected_paper_indices:
        new_citations.append([citing_paper_indx, cited_paper_idx])

new_citations = np.array(new_citations).T
new_edges = np.concatenate([edges, new_citations], axis=1)
```

In [35]:
```
print("Original node_features shape:", gnn_model.node_features.shape)
print("Original edges shape:", gnn_model.edges.shape)
gnn_model.node_features = new_node_features
gnn_model.edges = new_edges
gnn_model.edge_weights = tf.ones(shape=new_edges.shape[1])
print("New node_features shape:", gnn_model.node_features.shape)
print("New edges shape:", gnn_model.edges.shape)

logits = gnn_model.predict(tf.convert_to_tensor(new_node_indices))
probabilities = keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
Original node_features shape: (2708, 1433)
Original edges shape: (2, 5429)
New node_features shape: (2715, 1433)
New edges shape: (2, 5478)
Instance 1:
- Case_Based: 0.71%
- Genetic_Algorithms: 0.65%
- Neural_Networks: 78.55%
- Probabilistic_Methods: 2.99%
- Reinforcement_Learning: 0.55%
- Rule_Learning: 0.24%
- Theory: 16.31%
Instance 2:
- Case_Based: 0.06%
- Genetic_Algorithms: 98.13%
- Neural_Networks: 0.94%
- Probabilistic_Methods: 0.07%
- Reinforcement_Learning: 0.72%
- Rule_Learning: 0.04%
- Theory: 0.05%
Instance 3:
- Case_Based: 0.09%
- Genetic_Algorithms: 3.35%
- Neural_Networks: 37.28%
- Probabilistic_Methods: 52.64%
- Reinforcement_Learning: 0.9%
- Rule_Learning: 0.02%
- Theory: 5.72%
Instance 4:
- Case_Based: 0.02%
- Genetic_Algorithms: 0.11%
- Neural_Networks: 0.28%
- Probabilistic_Methods: 98.16%
- Reinforcement_Learning: 0.11%
- Rule_Learning: 0.0%
- Theory: 1.32%
Instance 5:
- Case_Based: 0.79%
- Genetic_Algorithms: 0.6%
- Neural_Networks: 1.03%
- Probabilistic_Methods: 0.32%
- Reinforcement_Learning: 20.71%
- Rule_Learning: 9.39%
- Theory: 67.17%
Instance 6:
```

```
- Case_Based: 0.38%
- Genetic_Algorithms: 1.23%
- Neural_Networks: 0.53%
- Probabilistic_Methods: 0.23%
- Reinforcement_Learning: 78.07%
- Rule_Learning: 3.34%
- Theory: 16.21%
Instance 7:
- Case_Based: 0.14%
- Genetic_Algorithms: 0.12%
- Neural_Networks: 0.42%
- Probabilistic_Methods: 0.4%
- Reinforcement_Learning: 2.65%
- Rule_Learning: 0.85%
- Theory: 95.42%
```

In [ ]: